# A New Elimination-Based Data Flow Analysis Framework Using Annotated Decomposition Trees

Bernhard Scholz[1] and Johann Blieberger[2]

[1] The University of Sydney
[2] Technische Universität Wien

**Abstract.** We introduce a new framework for elimination-based data flow analysis. We present a simple algorithm and a delayed algorithm that exhibit a worst-case complexity of $\mathcal{O}(n^2)$ and $\tilde{\mathcal{O}}(m)$. The algorithms use a new compact data structure for representing reducible flow graphs called *Annotated Decomposition Trees*. This data structure extends a binary tree to represent flowgraph information, dominance relation of flowgraphs, and the topological order of nodes. The construction of the annotated decomposition trees runs in $\mathcal{O}(n + m)$. Experiments were conducted with reducible flowgraphs of the SPEC2000 benchmark suite.

## 1 Introduction

Elimination-based approaches [19] are used for data flow analysis problems [17, 18, 15, 5, 3, 4] that cannot be solved with iterative approaches [12, 8]. There exist other applications for elimination methods, which go beyond the area of program analysis [24]. For solving data flow analysis problems there are two families of elimination-based approaches: algebraic methods and methods using path expressions.

Algebraic elimination methods [1, 9, 6, 21] consist of three steps: (1) reducing the flowgraph to a single node, (2) eliminating variables in the data flow equations by substitution, and (3) back-propagating the solution to other nodes. Algebraic elimination methods require two algebraic operations for a set of equations: *substitution* and *loop-breaking*. The substitution transformation is the replacement of the occurrence of a variable by its term whereas loop-breaking eliminates the occurrence of a variable on the right-hand side. Though not very efficient, Gaussian elimination is a generic algebraic elimination method to solve data flow equations in cubic time [16].

Path expressions were introduced in [24] to solve data flow equations. The flowgraph is seen as a deterministic finite state automata [10] whose language consists of all paths emanating from the start node to a node. The language is represented as a regular expression whose alphabet is the edge-set of the flowgraph. To find the data flow solution of a node, a path homomorphism is applied to the path expression. The operators $\cdot$, $\cup$, and $*$ of the regular expressions are re-interpreted. An elimination method using path expressions comprises two steps: (1) the computation of path expressions for all nodes in the flowgraph, and (2) the application of the path homomorphism. An inefficient algorithm for converting flowgraph to path expressions is described in [10] and runs in $\mathcal{O}(n^3)$.

In this paper we present a new path expression algorithm using the decomposition properties of reducible flowgraphs. The contribution of our work is threefold:

- a new representation of reducible flowgraphs called *Annotated Decomposition Trees* that combines control flow information, dominance relation, and topological order,
- an algorithm for computing annotated decomposition trees in linear time,
- an elimination framework based on annotated decomposition trees that computes path expressions.

The paper is organized as follows. In Section 2 we describe the basic notions required to present our approach. In Section 3 we outline the idea behind elimination-based methods using path expressions and show a motivating example. In Section 4 we present the construction of annotated decomposition trees for reducible flowgraphs. In Section 5 we show a simple and a delayed algorithm for computing path expressions. In Section 6 we present the results of our experiment. Related work is surveyed in Section 7. We draw our conclusions in Section 8.

## 2 Background

**Flowgraph and Path Expressions.** A *flowgraph* is a directed graph $G(V, E, r)$ where $V$ is the set of nodes and $E$ is the set of edges. We refer to $n$ as the number of nodes and $m$ as the number of edges. A flowgraph is trivial if there is a single node in $V$. Edge $u \rightarrow v$ has *source* $u$ and *destination* $v$. Vertex $r$ is a distinguished *root node* (aka. start node). A path $\pi$ is a sequence of edges $\langle(v_1 \rightarrow v_2), (v_2 \rightarrow v_3), \ldots, (v_{k-1} \rightarrow v_k)\rangle$ such that two consecutive edges $(v_i \rightarrow v_{i+1}) \in E$ and $(v_{i+1} \rightarrow v_{i+2}) \in E$ share the same node $v_{i+1}$. The empty path is denoted by $\varepsilon$. In a flowgraph all nodes are reachable, i.e., there is a path from $r$ to every other node in $V$.

**Definition 1.** *The path set Paths*$(u, v)$ *is the set of all paths from* $u$ *to* $v$ *in the flowgraph.*

In a regular expression, $\varepsilon$ denotes the empty string, $\emptyset$ denotes the empty set, $\cup$ denotes set union, $\cdot$ denotes concatenation, and $*$ denotes reflexive, transitive closure under concatenation. Thus each regular expression $R$ over $\Sigma$ represents a set $\sigma(R)$ of strings over an alphabet $\Sigma$ defined as:

1. $\sigma(\varepsilon) = \{\varepsilon\}$; $\sigma(\emptyset) = \emptyset$; $\sigma(a) = \{a\}$ for $a \in \Sigma$.
2. $\sigma(R_1 \cup R_2) = \sigma(R_1) \cup \sigma(R_2) = \{w \mid w \in \sigma(R_1)$ or $w \in \sigma(R_2)\}$;
3. $\sigma(R_1 \cdot R_2) = \sigma(R_1) \cdot \sigma(R_2) = \{w_1 w_2 \mid w_1 \in \sigma(R_1)$ and $w_2 \in \sigma(R_2)\}$;
4. $\sigma(R^*) = \bigcup_{k=0}^{\infty} \sigma(R)^k,$ where $\sigma(R)^0 = \{\varepsilon\}$ and $\sigma(R)^i = \sigma(R)^{i-1} \cdot \sigma(R)$.

For the algorithms in this paper we implicitely use simplifications for the regular expression operators: $[\varepsilon \cdot R] = R$, $[R \cdot \varepsilon] = R$, $[\emptyset \cdot R] = \emptyset$, $[R \cdot \emptyset] = \emptyset$, $[\emptyset \cup R] = R$, $[R \cup \emptyset] = R$, $[\emptyset^*] = \varepsilon$, and $[\varepsilon^*] = \varepsilon$.

**Definition 2.** *A* path expression $P(u, v)$ *is a regular expression over* $E$ *whose language* $\sigma(P(u, v))$ *is the path set Paths*$(u, v)$.

A node $u$ *dominates* a node $v$ if all paths from $r$ to $v$ include node $u$. All nodes $u$ that dominate $v$ are also called *dominators* of $v$. The immediate dominator $u$ of a node $v$ is a *dominator* of $v$ that does not dominate any other dominator of $v$ and $u$ is not $v$. The immediate dominator of node $v$ is written as *idom* $(v)$. The immediate dominators of nodes form a tree called a *dominator tree*.

A *back edge* in a flowgraph is an edge whose destination dominates its source. A flowgraph is *reducible* if the set of edges $E$ can be partitioned into disjoint sets $E_F$ and $E_B$ where $E_F$ is the set of *forward edges* and $E_B$ is the set of back edges. The set of forward edges must form a directed acyclic graph. A graph which is not reducible is called *irreducible*. Reducible flowgraphs (RFG) have the property that for each loop there exists a single entry point.

**Binary Leaf Trees.** A *binary leaf tree*[3] $T(V, E, r)$ is a rooted binary tree whose inner nodes always have two children. Set $V$ is the set of nodes, $E$ is the set of edges and $r$ is the root node of the tree. The left child of a node $x$ is denoted by $l(x)$ and the right child as $r(x)$. The parent node of a node $x$ is denoted by $p(x)$. A path in the tree is a sequence of nodes $\langle v_1, v_2, \ldots, v_k\rangle$ for which $v_i$ is the parent of $v_{i+1}$ for all $i$, $1 \leq i < k$.

**Data Flow Analysis.** A monotone data flow analysis problem [8] is a tuple DFAP($L, \wedge, F, c, G, M$), where $L$ is a bounded semi-lattice with meet operation $\wedge$, $F \subseteq L \rightarrow L$ is a monotone function space associated with $L$, $c \in L$ are the "data flow facts" associated with start node $r$, $G(V, E, r)$ is a flowgraph, and $M : E \rightarrow F$ is a map from $G$'s edges to data flow functions. We extend function $M$ to map a path $\pi = \langle(u_1 \rightarrow u_2), \ldots, (u_{k-1} \rightarrow u_k)\rangle$ to a function of $F$.

$$M(\pi) = \begin{cases} M(u_{k-1} \rightarrow u_k) \circ \ldots \circ M(u_1 \rightarrow u_2), & \text{if } \pi \neq \varepsilon \\ \iota, & \text{otherwise} \end{cases} \tag{1}$$

where $\iota$ is the identity function.

## 3 Motivation

In program analysis we compute a value for each node in the flowgraph. This value is called the meet-over-all-paths solution. It is the solution of applying the meet operator for the analysis result of all paths from the root node to a given node in the program[4].

**Definition 3.** *The meet-over-all-path (MOP) solution is defined for a node $u \in V$ as*

$$MOP(u) = \bigwedge_{\pi \in Paths(r,u)} M(\pi)(c) \tag{2}$$

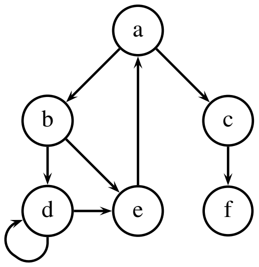
*where $c$ is the initial data flow value associated with the root node.*

Elimination methods directly compute the MOP solution. They solve the set of local data flow equations by either using substitution and elimination (aka. Gaussian Elimination) or by employing a path homomorphism [24]. In this work we focus on the latter approach. Path expressions represent path sets that are mapped into the function space of the data flow problem. This mapping is defined by reinterpreting the $\cup$, $\cdot$, and $*$ operators used to construct regular expressions as shown in [23]. The central idea of elimination methods is that MOP is computed as:

$$MOP(u) = \bigwedge_{\pi \in Paths(r,u)} M(\pi)(c) = M(P(r, u))(c), \tag{3}$$

[3] Binary leaf trees are sometimes also called *extended binary trees* [13].

[4] For backward problems we are interested in the set of reverse paths from the end node to a given node.



$$\begin{aligned}
P(a,a) &= (a \rightarrow b \cdot (b \rightarrow d \cdot d \rightarrow d^* \cdot d \rightarrow e \cup \\
&\qquad b \rightarrow e) \cdot e \rightarrow a)^* \\
P(a,b) &= P(a,a) \cdot a \rightarrow b \\
P(a,c) &= P(a,a) \cdot a \rightarrow c \\
P(a,d) &= P(a,b) \cdot b \rightarrow d \cdot d \rightarrow d^* \\
P(a,e) &= P(a,b) \cdot b \rightarrow e \cup P(a,d) \cdot d \rightarrow e \\
P(a,f) &= P(a,c) \cdot c \rightarrow f
\end{aligned}$$

(a) Flowgraph

(b) Regular Expressions

**Fig. 1.** Running Example.

where $P(r,u)$ is the path expression for path set *Paths*$(r,u)$. The mapping function is extended with the following operators of the regular expressions

$$\begin{aligned}
M(P_1 \cdot P_2) &= M(P_2) \circ M(P_1) \\
M(P_1 \cup P_2) &= M(P_1) \wedge M(P_2) \\
M(P^*) &= M(P)^* \\
M(\varepsilon) &= \iota
\end{aligned}$$

The operation $M(P_2) \circ M(P_1)$ is the function composition and $M(P)^*$ is a fixpoint operation. For simple data-flow analysis problems the fixpoint operation is quite often the identity function [19].

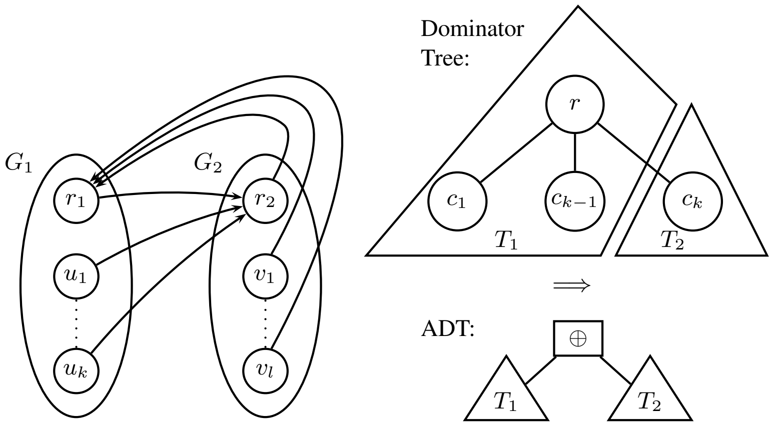
The complexity of elimination methods using path expressions depends on the path expression size. Consider the example depicted in Figure 1. Assume we want to solve a data flow analysis problem for the flowgraph given in Figure 1(a). An elimination method using path expressions computes a path expression as given in Figure 1(b). Then, mapping function $M$ is applied to path expression $P(r,u)$. To improve the performance of such an approach, expressions are reused (such as $P(a,a)$ in the running example). Without reuse of sub-expressions the memory complexity grows exponentially with the size of the flowgraph [10].

## 4 Annotated Decomposition Trees

For the elimination framework we introduce a new data structure called an Annotated Decomposition Tree (ADT) that recursively splits the reducible flowgraph into intervals. An interval is a subgraph of the flowgraph and has the following properties: (1) every interval has a single entry node, and (2) the single-entry node of the interval dominates all nodes of the interval.

The ADT is a binary leaf tree. An inner node in the ADT represents a composition operation that composes two disjoint intervals $G_1$ and $G_2$. The leaves of the tree represent trivial intervals consisting of a single node in the flowgraph[5]. The composition operation is a generalisation of work published in [25, 26, 11].

[5] Because ADTs are binary leaf trees, there are $n-1$ inner nodes where $n$ is the num. of leaves.



(a) Composition

(b) Decomposition

**Fig. 2.** Composition and Decomposition of Reducible Flowgraphs.

**Definition 4.** *Let $G_1(V_1, E_1, r_1)$ and $G_2(V_2, E_2, r_2)$ be flowgraphs such that $V_1$ and $V_2$ are disjoint sets. The composition $G_1 \oplus_{(F,B)} G_2$ is defined as*

$$(V_1 \cup V_2, E_1 \cup E_2 \cup (F \times \{r_2\}) \cup (B \times \{r_1\}), r_1)$$

*where $F \subseteq V_1$ and $B \subseteq V_2$ denote the sources of the forward and backward edges. Node $r_1$ becomes the new single-entry node of the composed interval.*

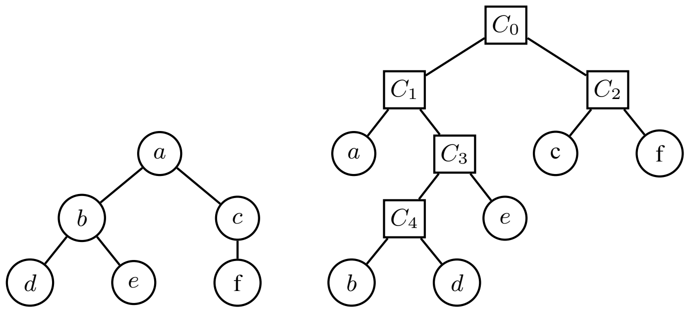
The composition of two intervals $G_1$ and $G_2$ is depicted in Figure 2(a). The single-entry nodes of the intervals are denoted by $r_1$ and $r_2$. The edge set $F \times \{r_2\}$ connects a subset of nodes in $G_1$ to $r_2$. The edge set $B \times \{r_1\}$ connects a subset of nodes in $G_2$ to $r_1$.

By Definition 4, root node $r_1$ dominates all nodes of $G_1$ and $G_2$ because every node in the composed interval can only be reached via $r_1$. The same holds for $r_2$, i.e., $r_2$ dominates all nodes in $G_2$. This implies that the nodes of $G_1$ form a sub-tree in the dominator tree with $r_1$ as a root-vertex of the sub-tree, and single-entry node $r_2$ is immediately dominated by $r_1$.

The forward edges of a reducible flow graph form a directed acyclic graph imposing a topological order $<$ such that for all edges $(u, v) \in E_F$, $u < v$ holds. Since the single-entry node of an interval dominates all nodes in the interval, the single-entry node of the interval is smaller than the nodes in the interval with respect to the topological order. The composition implies that $r_1 < r_2$. Given a composition $G_1 \oplus G_2$, the inequality

$$\forall u \in V_1 : \forall v \in V_2 : r_1 \leq u < r_2 \leq v \tag{4}$$

holds. Assume a total order $R$ of nodes in the flowgraph $[u_1, \ldots, u_n]$ such that for $(u_i, u_j) \in E_F$, $i < j$. An interval decomposition of the flowgraph partitions the ordered nodes into two parts. Vertex $r_1$ has index 1 and all the nodes between 1 and $r_2 - 1$ belong to the interval $G_1$. The nodes from $r_2$ to $n$ belong to $G_2$. By recursively applying the decomposition for ordered nodes, we have a range representation of the tree. For example a possible total order for the flowgraph in Figure 1(a) is $[a, b, d, e, c, f]$. The first composition of the ADT splits the ordered nodes in two halves, i.e., $[[a, b, d, e], [c, f]]$.



(a) Ordered Dominator Tree (b) Decomposition Tree

**Fig. 3.** Dominator and Decomposition Tree of Example.

By recursively splitting intervals, we obtain $[[[a],[[b,d],e]],[c,f]]$ representing the intervals of the flowgraph.

In the following we deal with the problem of finding an ADT for a given flowgraph. Because there might be several possible topological orders of a flowgraph, we can have several ADTs for a single flowgraph. However, for a given topological order of a flowgraph there exists a single ADT. The ADT is constructed by using the dominator tree and a given topological order.

We observe that the root node $r_2$ is immediately dominated by $r_1$ and therefore is a child of $r_1$ in the dominator tree. Assume that the children $c_1,\ldots,c_{k-1},c_k$ of node $r_1$ in the dominator tree are ordered by the topological order, i.e., $c_i < c_j$. Vertex $r_2$ is the child $c_k$ (cf. Equation 4) and the nodes of $G_2$ are nodes which are dominated by $r_2$. A simple decomposition scheme of the ordered dominator tree (as illustrated in Figure 2(b)) allows the construction of the ADT. Interval $G_1$ is the result of the decomposition of the dominator tree without subtree $c_k$. Interval $G_2$ is the result of the decomposition of subtree $c_k$.

The decomposition of the dominator tree results in a simple algorithm for constructing an ADT: (1) order the children of the dominator tree with respect to topological order of the nodes and (2) recursively traverse the ordered dominator tree and construct the ADT. The algorithm for constructing the decomposition tree is shown in Figure 4. For constructing an order among children we use a stack $s_u$ for each node $u$ in the flowgraph. Procedure *ConstructADT* pushes nodes in reverse topological order onto the stack of its immediate dominator. Before calling *Traverse* in Procedure *ConstructADT*, the stack of a node contains all its children in reverse topological order. The element on top of the stack is the right-most child of the node and the bottom element of the stack is the left-most child of the node. The stack allows us to partition the graph as illustrated in Figure 2(b).

The construction of the ADT is performed in function *Traverse*. Function *Leaf* with parameter $u$ creates a new leaf in the decomposition tree where $u$ is a node of the flowgraph. Function *Node* creates an inner node with a left and right child. The construction is performed recursively beginning with the root node of the dominator tree. Inside the loop the children of node $u$ are popped from the stack in reverse topological order and for each child a decomposition operation is created. The function *Traverse* pops exactly

```
CONSTRUCTADT ()
for $i \leftarrow |V| \ldots 2$ do
    $u \leftarrow$ order$(i)$
    $v \leftarrow$ idom $(u)$
    PUSH $u$ onto $s_v$
endfor
return TRAVERSE$(r)$
```

```
TRAVERSE $(u)$
$x \leftarrow$ LEAF$(u)$
while stack $s_u$ is not empty do
    $v \leftarrow$ POP node from $s_u$
    $x \leftarrow$ NODE$(x,$ TRAVERSE$(v))$
endwhile
return $x$
```

**Fig. 4.** Construction of the ADT.

$n - 1$ elements from the node stacks. We have $n$ function calls of *Traverse*. Thus, the space and time complexity of function *Traverse* is $\mathcal{O}(n)$.

For the running exampe the dominator tree is shown in Figure 3(a). The children of the nodes are ordered with respect to the topological order. The first composition is the cut between node $a$ and $c$ because $c$ is the right-most children with respect to the topological order. The resulting two dominator trees are recursively cut and each cut represents an inner node in the ADT. The resulting decomposition tree is depicted in Figure 3(b).

So far, we have not discussed how to determine the forward and backward edges of a composition. To compute $F$- and $B$-sets we traverse the set of edges and associate each edge to a composition in the ADT. The edge is associated to the composition node in the ADT that is the nearest common ancestor of leaves $u$ and $v$. An edge is an element of set $F$ if it is a forward edge, otherwise it is element of set $B$. Set $F$ is empty for leaf nodes in the ADT. The algorithm in Figure 5(a) annotates the decomposition tree with sets $F$ and $B$. It exhibits a complexity of $\mathcal{O}(n + m)$ by using the efficient nearest common ancestor algorithms [7, 2] with a complexity of $\mathcal{O}(1)$ for a single NCA query.

The $F$- and $B$-sets are given in Figure 5(b). For example, consider edge $a \rightarrow c$. The nearest common ancestor of leaves $a$ and $c$ is the extended composition $C_0$ in the decomposition Tree of Figure 3(b). Vertex $a$ is smaller then node $c$ in the topological order. Therefore, the edge is a forward edge and stored in $F_{C_0}$. Because the target of an edge is inherently defined by the composition, i.e. either node $r_1$ or node $r_2$ depending whether it is a forward or backward edge, we only add the source of an edge to $F$ or $B$. This means, that for edge $a \rightarrow c$ node $a$ is added to $F_{C_0}$.

## 5 Path Expressions

We compute path expressions for nodes using the annotated decomposition tree of a reducible flowgraph as the underlying data structure. Path expressions are computed by an inductive scheme. For the inductive step we construct path expressions by using the properties of the composition (see Def. 4).

**Theorem 1.** *If $G(V, E, r)$ is a trivial flowgraph, then*

$$P(r, r) = \begin{cases} (r \rightarrow r)^*, & \text{if } (r \rightarrow r) \in E, \\ \varepsilon, & \text{otherwise.} \end{cases} \tag{5}$$

*Otherwise the flowgraph is composed and $G(V, E, r) = G_1(V_1, E_1, r_1) \oplus_{(F,B)} G_2(V_2, E_2, r_2)$. For given path expressions $P_1(r_1, u)$ of $G_1$ (for all $u \in V_1$) and given path*

```
ComputeFBSets($ADT$)
for all $u \to v \in E$ do
    $x \leftarrow \textsc{nca}(ADT, u, v)$
    if $u \to v \in E_B$ then
        $B_x \leftarrow B_x \cup \{u\}$
    else
        $F_x \leftarrow F_x \cup \{u\}$
    endif
endfor
```

(a) Algorithm

| $u$ | $F_u$ | $B_u$ |
|---|---|---|
| $C_0$ | $\{a\}$ | $\{\}$ |
| $C_1$ | $\{a\}$ | $\{e\}$ |
| $C_2$ | $\{c\}$ | $\{\}$ |
| $C_3$ | $\{b, d\}$ | $\{\}$ |
| $C_4$ | $\{b\}$ | $\{\}$ |
| $a$ | n/a | $\{\}$ |
| $b$ | n/a | $\{\}$ |
| $c$ | n/a | $\{\}$ |
| $d$ | n/a | $\{d\}$ |
| $e$ | n/a | $\{\}$ |
| $f$ | n/a | $\{\}$ |

(b) F- and B-sets

**Fig. 5.** Algorithm and Example for Computing F- and B-Sets.

*expressions $P_2(r_2, v)$ of $G_2$ (for all $v \in V_2$), the path expressions of the composed flowgraph are:*

$$\forall u \in V_1 : P(r, u) = L \cdot P_1(r_1, u) \tag{6}$$

$$\forall v \in V_2 : P(r, v) = R \cdot P_2(r_2, v) \tag{7}$$

*where*[6]

$$X = \bigcup_{u \in F} P_1(r_1, u) \cdot (u \to r_2) \tag{8}$$

$$Y = \bigcup_{v \in B} P_2(r_2, v) \cdot (v \to r_1) \tag{9}$$

$$L = [X \cdot Y]^* \tag{10}$$

$$R = L \cdot X \tag{11}$$

*Proof.* See Appendix.

*Simple Algorithm:* A simple algorithm traverses the ADT in bottom-up fashion and updates the path expressions for nodes in $G_1$ and $G_2$ according to Theorem 1. The complexity of the simple algorithm is $\mathcal{O}(n^2)$ because in the worst case $\mathcal{O}(n)$ updates are performed for a node in the ADT and there are $\mathcal{O}(n)$ nodes in the ADT.

*Delayed Algorithm:* A more efficient algorithm does not update the path expressions of all nodes in the intervals $G_1$ and $G_2$. Only the nodes in $F$ and $B$ of a composition are updated and the update of the remaining nodes is deferred to a later stage.

The construction of path expressions implies that a path expression of node $u$ in the flowgraph is a sequence of $L$ and $R$ prefixes followed by either $\varepsilon$ or $(u \to u)^*$. We store the path expressions $L$ and $R$ at the left and right child of an inner node in the ADT and the path expression $\varepsilon$ or $(u \to u)^*$ at the leaves. Then, a path from the root of the ADT to node $u$ defines the path expression[7] by mapping the nodes of the path to their path expressions. This observation enables the usage of a path compression scheme [22] to implement the delayed update.

[6] $\bigcup_{x \in X} f(x)$ is the empty set if set $X$ is empty.

[7] Without loss of generality we store $\varepsilon$ in the root of the ADT.

```
COMPUTEPATHEXPR ($w$)
if $w$ is not a leaf then
    COMPUTEPATHEXPR($l(w)$)
    COMPUTEPATHEXPR($r(w)$)
    $X \leftarrow \bigcup_{u \in F_w} [\mathrm{EVAL}(u) \cdot u \rightarrow r_2]$
    $Y \leftarrow \bigcup_{v \in B_w} [\mathrm{EVAL}(v) \cdot v \rightarrow r_1]$
    $L \leftarrow [X \cdot Y]^*$
    $R \leftarrow L \cdot X$
    $z \leftarrow l(w)$
    if $z$ is a leaf and $z \rightarrow z \in E$ then
        $L \leftarrow L \cdot (z \rightarrow z)^*$
    endif
    $z \leftarrow r(w)$
    if $z$ is a leaf and $z \rightarrow z \in E$ then
        $R \leftarrow R \cdot (z \rightarrow z)^*$
    endif
    LNK_UPD($w, r(w), L$)
    LNK_UPD($w, l(w), R$)
endif
```

```
LNK_UPD ($x, y, v$)
$p(y) \leftarrow x$
$R_y \leftarrow v$
```

```
EVAL ($x$)
if $p(x) \neq p(p(x))$ then
    $R_x \leftarrow [\mathrm{EVAL}(p(x)) \cdot R_x]$
    $p(x) \leftarrow p(p(x))$
endif
return $R_x$
```

```
MAIN ()
$adt \leftarrow$ CONSTRUCTADT()
COMPUTEFBSETS($adt$)
COMPUTEPATHEXPR($adt$)
if $G$ is not trivial then
    for $u \in V$ do
        $P(r, u) \leftarrow \mathrm{EVAL}(u)$
    endfor
else
    $P(r, r) \leftarrow \begin{cases} (r \rightarrow r)^*, & \text{if } (r \rightarrow r) \in E, \\ \varepsilon, & \text{otherwise.} \end{cases}$
endif
```

**Fig. 6.** Computing Path Expressions: Delayed Algorithm.

In Figure 6 we outline the algorithm for constructing path expressions with a delayed update. The LNK_UPD operation assigns a path expression to a node in the ADT and constructs the tree for path compression. The EVAL operation queries the sequence of $L$ and $R$ prefixes for a node of the flowgraph. The complexity of the algorithm is bounded by the number of composition nodes in the decomposition tree (i.e. $n - 1$) and the number of leaves (i.e. $n$). With a simple path compression scheme the delayed update is bounded by $\mathcal{O}(m \log n)$ (as outlined in Figure 6). A more sophisticated path compression algorithm with $\mathcal{O}(m\alpha(m, n))$ introduced in [22] improves the upper bound of the overall algorithm. However, in practice the sophisticated path compression scheme will not be superior to the simple path compression scheme due to small problem sizes [14].

## 6 Experimental Results

We have implemented the simple and the delayed algorithm in C and measured the performance of the algorithms on a 2.6GhZ AMD computer. We also implemented Sreedhar's algorithm [21] and compared its performance to the simple and the delayed algorithm. As a benchmark we used the SPEC2000 benchmark suite. The flowgraphs were generated by the GCC compiler. In this experiment we are interested in the following numbers: (1) the execution time to construct ADTs for SPEC2000, (2) the speed-ups of the simple and delayed algorithm vs. Sreedhar's algorithm, and (3) the reduction of the number of $\cdot$, $\cup$ and $*$ operators by using the delayed algorithm.

The results of the experiment are shown in Table 1. The execution times for constructing the ADT and computing the path expressions are given in columns $t_{adt}$ and $t_c$. Note that all time measurements are in microseconds. The construction of ADTs

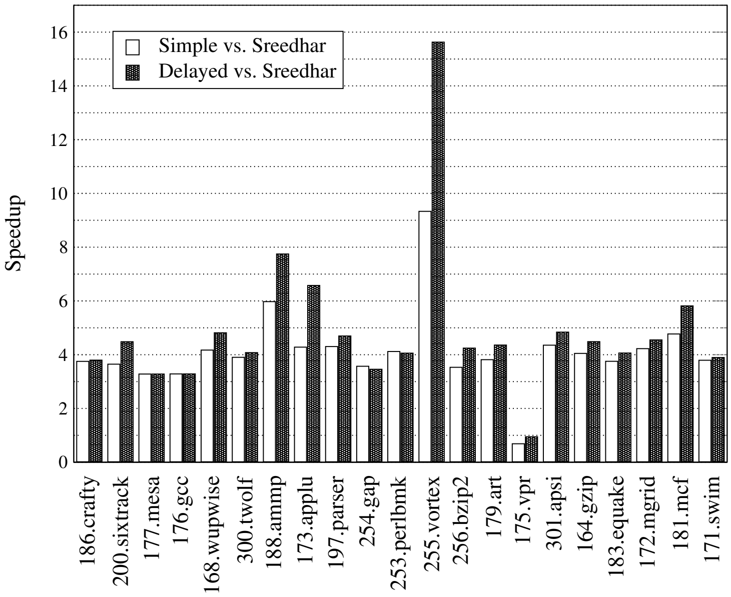


**Fig. 7.** Speed-Ups: Simple and Delayed Algorithm vs. Sreedhar.

is fast but it takes longer than computing path expressions. The computation of ADTs requires four steps: (1) compute topological order, (2) compute dominator tree, (3) perform the pre-processing step for NCAs, and (4) construct the ADT using the dominator tree, topological order, and NCA relation. Hence, the execution time to construct ADTs has the same magnitude as computing path expressions with the simple algorithm. The delayed algorithm has a significantly smaller runtime and is approx. 1.8 times faster than the simple algorithm.

We compared the runtime of our algorithms[8] with the runtime of a C++/STL implementation of Sreedhar's eager algorithm. For the comparison we measured the time to construct the DJ-graphs (Column $t_{dj}$) and the reduction phase ($t_r$). However, we did not measure the propagation phase which is a simple traversal over the dominator tree. The speed-ups of the simple and delayed algorithm vs. Sreedhar's algorithm are shown for each Spec2000 benchmark in Figure 7. The speed-ups vary depending on the size of the flowgraphs. For small flowgraphs the execution time of the simple and delayed algorithm is of the same magnitude. For large flowgraphs the execution time diverges by up to a factor of 9.3 and 15.6, respectively.

The delayed algorithm has significantly smaller path expressions. The delayed algorithm reduces the number of regular expression operators by 38.1%. We attribute the more compact path expressions to the re-use of regular expressions. Though the number of $*$ and $\cup$ operators is the same for both algorithms, the number of $\cdot$ operators is reduced by a factor of three. This substantial reduction is due to reusing the same $L$ and $R$ sub-sequences. The reductions for all benchmarks are shown in Figure 8. The reductions vary between 24.8% (best case) and 83.0% (worst case) depending on how many

[8] Our algorithms are highly-tuned C-algorithms.

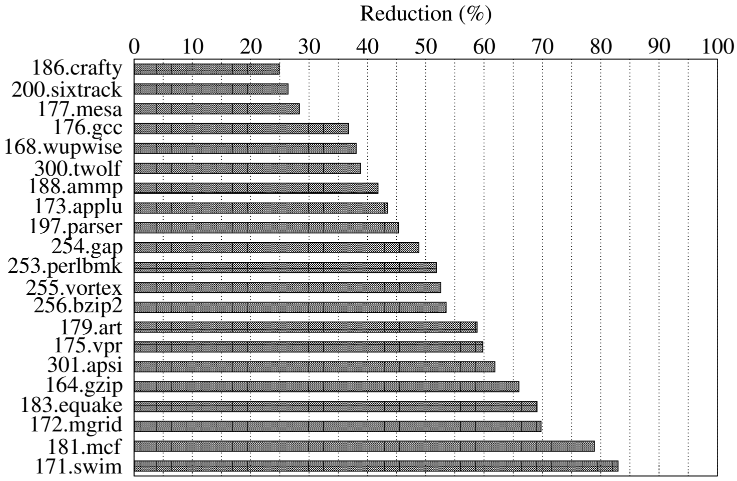


**Fig. 8.** Reductions of Operations (%) between 0% and 100%. Benchmarks achieving better reductions are listed first.

sub-sequences of $L$ and $R$ prefixes can be reused. Larger flowgraphs have a greater potential for reuse of sub-path expressions.

## 7 Discussion and Related Work

Besides Gaussian elimination with order $\mathcal{O}(n^3)$ complexity, there are five elimination algorithms known in literature: (1) *Allen-Cocke interval analysis* [1], (2) *Hecht-Ullman $T_1 - T_2$ analysis* [9], (3) *Graham-Wegman analysis* [6], (4) *Sreedhar-Gao-Lee DJ graph based analysis* [21], (5) *Tarjan interval analysis* [23], (see [19] for a comparison of the first four algorithms). Algorithms (1) to (4) are algebraic elimination-based algorithms using substitution and loop-breaking. Only (5) is an elimination-based approach using path expressions. Note the approach introduced in [20] is marginally related to elimination-based algorithms because this approach detects standard structured control-flow patterns, such as “if-then-else”, “begin-end”, or “while-do”, which is not our concern.

Allen-Cocke interval analysis establishes a natural partition of the variables and a variable order on each of a sequence of systems that, when used to order the equations, results in a highly structured coefficient matrix facilitating the equation-reduction process. Hecht-Ullman $T_1 - T_2$ analysis, Tarjan interval analysis, and Graham-Wegman analysis avoid repeated calculations of common substitution sequences in the equations by delaying certain computations. Sreedhar-Gao-Lee DJ graph based analysis employs structural information of the so-called DJ graph, a union of the CFG and its dominator tree, to find efficient substitution sequences.

Hecht-Ullman $T_1 - T_2$ analysis uses a nondeterministic substitution order for terms in the equations; the substitutions are recorded in a height-balanced $2-3$ tree to take advantage of possible common factors in subsequent calculations. Tarjan interval analysis establishes a linear variable order and eliminates variables from the system of equations in that order, delaying some calculations; a path compressed tree is used to remember

| Problemsize | | | | Simple Algo | | | | Delayed Algo | | Sreedhar's Algo | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bnchm. | n | m | $t_{\text{adt}}$ | $n_*$ | $n_\cup$ | $n.$ | $t_c$ | $n.$ | $t_c$ | $n_{E1}$ | $n_{E2a}$ | $n_{E2b}$ | $t_{dj}$ | $t_r$ |
| gzip | 1639 | 2369 | 2.09 | 201 | 613 | 5723 | 1.27 | 3500 | 1.23 | 199 | 636 | 966 | 5.36 | 7.25 |
| wupwise | 444 | 657 | 0.78 | 58 | 175 | 3031 | 0.72 | 1010 | 0.44 | 58 | 148 | 317 | 1.58 | 3.89 |
| swim | 109 | 158 | 0.16 | 27 | 28 | 280 | 0.09 | 223 | 0.09 | 27 | 30 | 56 | 0.43 | 0.39 |
| mgrid | 179 | 269 | 0.26 | 51 | 49 | 628 | 0.16 | 408 | 0.16 | 51 | 46 | 111 | 0.64 | 0.74 |
| applu | 590 | 899 | 0.78 | 172 | 153 | 3958 | 0.72 | 1538 | 0.52 | 172 | 156 | 567 | 1.93 | 4.33 |
| vpr | 4227 | 5973 | 5.51 | 498 | 1509 | 16672 | 3.54 | 9161 | 3.15 | 494 | 1437 | 2991 | 13.75 | 21.57 |
| gcc | 60818 | 96156 | 88.47 | 3412 | 33822 | 468454 | 90.28 | 148878 | 49.38 | 3303 | 24761 | 58003 | 203.85 | 863.93 |
| mesa | 21981 | 31418 | 32.22 | 1330 | 9171 | 196921 | 42.15 | 48257 | 16.20 | 1300 | 8071 | 17989 | 72.61 | 245.83 |
| art | 615 | 924 | 0.87 | 130 | 205 | 2720 | 0.68 | 1462 | 0.55 | 125 | 213 | 505 | 2.06 | 4.61 |
| mcf | 449 | 660 | 0.59 | 57 | 180 | 1290 | 0.34 | 968 | 0.37 | 57 | 173 | 265 | 1.53 | 1.79 |
| equake | 309 | 423 | 0.43 | 67 | 73 | 1041 | 0.24 | 676 | 0.25 | 66 | 66 | 232 | 1.10 | 1.66 |
| crafty | 6161 | 9563 | 7.69 | 403 | 3107 | 70633 | 14.13 | 14941 | 5.34 | 401 | 2359 | 6619 | 20.40 | 183.23 |
| ammp | 3773 | 5754 | 6.53 | 431 | 1725 | 24543 | 5.03 | 9013 | 3.09 | 427 | 1408 | 3390 | 12.30 | 28.53 |
| parser | 4967 | 7428 | 6.90 | 655 | 2099 | 27705 | 5.37 | 11056 | 3.84 | 652 | 1756 | 3584 | 16.16 | 30.65 |
| sixtrack | 6998 | 10576 | 15.41 | 947 | 2844 | 74721 | 13.57 | 16941 | 5.61 | 293 | 694 | 1619 | 6.83 | 13.07 |
| perlbmk | 6904 | 10464 | 9.11 | 265 | 3529 | 33429 | 6.89 | 15495 | 5.28 | 254 | 3004 | 4892 | 22.68 | 46.98 |
| gap | 21178 | 31685 | 26.78 | 1490 | 9847 | 109902 | 20.61 | 47878 | 15.98 | 1459 | 9188 | 16587 | 68.75 | 123.09 |
| vortex | 18633 | 27490 | 24.82 | 236 | 9539 | 80959 | 17.02 | 37950 | 13.86 | 235 | 7630 | 11138 | 60.62 | 96.50 |
| bzip2 | 1648 | 2495 | 2.07 | 277 | 642 | 7563 | 1.56 | 3619 | 1.30 | 276 | 590 | 1024 | 5.35 | 9.99 |
| twolf | 7294 | 11216 | 10.39 | 875 | 3227 | 51216 | 9.27 | 17397 | 5.74 | 863 | 2763 | 5845 | 23.76 | 70.05 |
| apsi | 2009 | 2968 | 2.64 | 367 | 688 | 7782 | 1.64 | 4413 | 1.53 | 364 | 696 | 1257 | 6.45 | 9.79 |
| Total | 170925 | 259545 | 244.50 | 11949 | 83225 | 1189171 | 235.28 | 394784 | 133.91 | 11076 | 65825 | 137957 | 548.15 | 1767.87 |

**Table 1.** Results of the simple and delayed ADT algorithm, and results of Sreedhar's algorithm. Columns $n$ and $m$ are the number of nodes and edges in the flowgraph. Column $t_{\text{ADT}}$ is the execution time to construct the ADT. $n_*$, $n_\cup$, and $n.$ are the number of regular expression operators and $t_c$ is the execution time to compute path expressions. Note that the $n_*$ and $n_\cup$ are identical for the simple and delayed algorithm. For Sreedhar's algorithm the number of E1, E2a, and E2b reductions are $n_{E1}$, $n_{E2a}$, and $n_{E2b}$. The execution time to construct the DJ-graph and the time to perform the reductions are $t_{dj}$ and $t_r$. All execution times are given in $\mu s$.

sequences of reduced equations for these delayed calculations. Graham-Wegman analysis establishes an order of substitutions for each term in the system that avoids duplication of common substitution sequence calculations. It uses a transformed version of the original flowgraph to remember previous substitutions. By delaying computations, Sreedhar-Gao-Lee DJ graph based analysis can be made more efficient. In contrast to Tarjan interval analysis, the Sreedhar-Gao-Lee algorithm employs simple path compression on the dominator tree.

Among the known elimination algorithms the best in terms of worst-case complexity is Tarjan's interval analysis algorithm, which balances the path compressed tree in a preprocessing operation. This algorithm has a runtime of $\mathcal{O}(m \log m)$ employing a simple path compression scheme. By using a sophisticated data structure for path compression a better upper bound of $\mathcal{O}(m\alpha(m, n))$ can be achieved. However, the simple path compression scheme will outperform the sophisticated one for typical problem sizes in program analysis.

Our algorithm is based on structural information of the decomposition tree. Thus it is more similar to the Sreedhar-Gao-Lee algorithm than to the other algorithms. It uses simple path compression employed on the decomposition tree to remember sequences of reduced equations for delayed calculations. It is, however, easy to use Tarjan's preprocessing operation and a separate data structure to achieve a more efficient version of our algorithm.

# 8 Conclusion

In this paper we introduced a new framework for elimination-based data flow analysis using path expressions. Elimination-based frameworks are used for program analysis problems [17, 18, 15, 5, 3, 4] that cannot be solved with iterative solvers. The framework uses a new data structure called annotated decomposition trees (ADTs) that comprises topological order, dominance relation, and the control flow. We presented a simple algorithm and a delayed algorithm that employed annotated decomposition trees as a data structure. The worst-case complexities of both algorithms are $\mathcal{O}(n^2)$ and $\tilde{\mathcal{O}}(m)$.

We conducted experiments with the SPEC2000 benchmark suite. The delayed algorithm runs 1.8 times faster than the simple algorithm and has 38.1% of the operators in comparison with the simple algorithm.

## Acknowledgement

We would like to thank Bernd Burgstaller, Shirley Goldrei, and Wei-ying Ho for their useful comments and for proof-reading the manuscript. This work has been partially supported by the ARC Discovery Project Grant “Compilation Techniques for Embedded Systems” under Contract DP 0560190.

## References

1. F. E. Allen and J. Cocke. A program data flow analysis procedure. *Comm. ACM*, 19(3):137–147, 1976.
2. M. Bender and M. Farach-Colton. The lca problem revisited. In *Proc. of Latin American Theoretical Informatics*, pages 88–94, 2000.
3. J. Blieberger. Data-flow frameworks for worst-case execution time analysis. *Real-Time Syst.*, 22(3):183–227, 2002.
4. R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant computations. In *Proc. of PLDI*, pages 1–14, 1998.
5. T. Fahringer and B. Scholz. A Unified Symbolic Evaluation Framework for Parallelizing Compilers. *IEEE TPDS*, 11(11), November 2000.
6. S. L. Graham and M. Wegman. Fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, 1976.
7. D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *Siam J. Comput.*, 13(2):338–355, May 1984.
8. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, New York, 1 edition, 1977.
9. M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Comput.*, 4(4):519–532, 1977.
10. J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, 2001.
11. R. Joshi, U. Khedker, V. Kakade, and M. Trivedi. Some interesting results about applications of graphs in compilers. *CSI J.*, 31(4), 2002.
12. G. A. Kildall. A unified approach to global program optimization. In *Proc. of Symposium on Principles of Programming Languages*, pages 194–206. ACM, ACM Press, 1973.
13. D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., third edition, 1997.
14. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

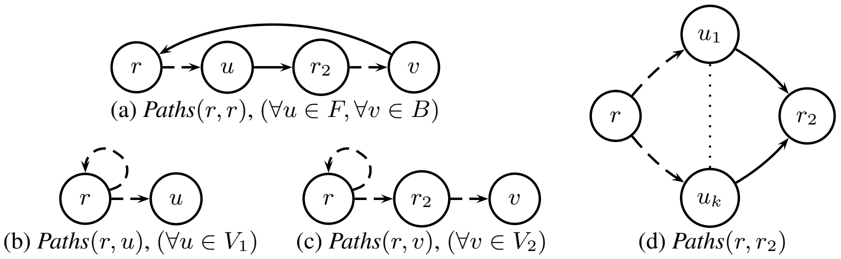

$r$
$u$
$r_2$
$v$
(a) *Paths*$(r, r)$, $(\forall u \in F, \forall v \in B)$
$r$
$u$
(b) *Paths*$(r, u)$, $(\forall u \in V_1)$
$r$
$r_2$
$v$
(c) *Paths*$(r, v)$, $(\forall v \in V_2)$
$u_1$
$r$
$r_2$
$u_k$
(d) *Paths*$(r, r_2)$

**Fig. 9.** Piecewise description of path sets: dotted lines are paths; solid lines are edges.

15. E. Mehofer and B. Scholz. A Novel Probabilistic Data Flow Framework. In *Proc. of CC*, pages 37 – 51, Genova, Italy, April 2001. Springer.
16. M. C. Paull. *Algorithm design: a recursion transformation framework*. Wiley-Interscience, New York, NY, USA, 1988.
17. G. Ramalingam. Data flow frequency analysis. In *Proc. of PLDI*, pages 267–277, New York, NY, USA, 1996. ACM Press.
18. T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *Proc. of ICSE '02*, pages 478–488, New York, NY, USA, 2002. ACM Press.
19. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–315, Sept. 1986.
20. M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.
21. V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM TOPLAS*, 20(2):388–435, 1998.
22. R. Tarjan. Applications of path compression on balanced trees. *J. of the ACM*, 26(4):690–715, Oct. 1979.
23. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.
24. R. E. Tarjan. A unified approach to path programs. *J. ACM.*, 28(3):577–593, 1981.
25. O. Vernet and L. Markenzon. Maximal reducible flowgraphs. Technical Report RT029/DE9, D. de Engenharia de Sistemas, Instituto Militar de Engenharia, Rio de Janeiro, Brasil, 1998.
26. O. Vernet and L. Markenzon. Solving problems for maximal reducible flowgraphs. *Disc. Appl. Math.*, 136:341–348, 2004.

# A Appendix

**Definition 5.** *Path* $a = \langle (u_1, u_2), \ldots, (u_{k-1}, u_k) \rangle$ *is connectable to path* $b = \langle (v_1, v_2), \ldots, (v_{l-1}, v_l) \rangle$ *if* $u_k$ *is equal to* $v_1$.

**Definition 6.**

$$A \cdot B = \begin{cases} \bigcup_{a \in A} \bigcup_{b \in B} a \cdot b, & \textit{if } a \textit{ is connectable to } b \\ \emptyset, & \textit{otherwise} \end{cases} \tag{12}$$

*Proof.* (Proof of Theorem 1). Proof by structural induction over the ADT. Each subtree in the ADT represents a sub-flowgraph of the flowgraph.
*Inductive Hypothesis:* $\forall u \in V : \sigma(P(r, u)) = \textit{Paths}(r, u)$
*Basis:* Both cases are trivially true by Proposition of Theorem 1.

*Induction Step*: The composition operation $\oplus_{(F,B)}$ (cf. Figure 2 and Def. 4) allows us to break the paths of the composition into segments, as suggested in Figure 9. Note that $r_1$ becomes $r$ after the composition. Path set $\mathit{Paths}(r, r)$ is depicted in Figure 9(a). A path in $\mathit{Paths}(r, r)$ starts in $r$ and uses a path in $G_1$ as a sub-path to reach a node $u \in F$. From node $u \in F$ there exists an edge to node $r_2$ by Def. 4. From node $r_2$ a path in $G_2$ is used as a sub-path to reach a node $v \in B$. By Def. 4 there exists an edge from $v \in B$ to $r$. Since a path may consist of several cycles we express the path set $\mathit{Paths}(r, r)$ as

$$\mathit{Paths}(r,r) = \bigcup_{i\geq 0}\left[\bigcup_{u\in F}\bigcup_{v\in B}\mathit{Paths}(r,u)\cdot\{\langle u\rightarrow r_2\rangle\}\cdot\mathit{Paths}(r_2,v)\cdot\{\langle v\rightarrow r\rangle\}\right]^i \tag{13}$$

where the inner term describes for a concrete $u \in F$ and $v \in B$ all possible simple cycles. Note that if set $B$ is empty, the path set $\mathit{Paths}(r, r)$ becomes $\varepsilon$ because the inner term reduces to an empty set and Kleene's closure of the empty set yields $\varepsilon$, i.e. $\emptyset^0 = \varepsilon$. A node $u \in V_1$ is described by path set $\mathit{Paths}(r, r)$ concatenated by path set $\mathit{Paths}(r_1, u)$ that is a path in $G_1$ as illustrated in Figure 9(b). Therefore,

$$\forall u \in V_1 : \mathit{Paths}(r,u) = \mathit{Paths}(r,r)\cdot\mathit{Paths}(r_1,u). \tag{14}$$

As depicted in Figure 9(c), a node $v \in V_2$ can be described by the concatenation of a path from $r$ to $r_2$ and a path in $G_2$ from $r_2$ to node $v$:

$$\forall v \in V_2 : \mathit{Paths}(r,v) = \mathit{Paths}(r,r)\cdot\mathit{Paths}(r,r_2)\cdot\mathit{Paths}(r_2,v) \tag{15}$$

The paths of path set $\mathit{Paths}(r, r_2)$ are depicted in Figure 9(d). The possible paths from $\mathit{Paths}(r_1, u)$ to $r_2$ are merged:

$$\mathit{Paths}(r,r_2) = \bigcup_{u\in F}\mathit{Paths}(r_1,u)\cdot\{\langle u,r_2\rangle\} \tag{16}$$

It can be shown by an indirect argument (using Def. 4) that all paths from $r$ to $u \in V_1$ are contained in set $\mathit{Paths}(r, u)$ of Equation 14 and that all paths from $r$ to $v \in V_2$ are contained in set $\mathit{Paths}(r, v)$ of Equation 15. By using the inductive hypothesis we transform Equation 13 to the following path expression:

$$\mathit{Paths}(r,r) = \bigcup_{i\geq 0}\left[\left(\bigcup_{u\in F}\sigma(P_1(r_1,u)\cdot(u\rightarrow r_2))\right)\cdot\left(\bigcup_{v\in B}\sigma(P_2(r_2,v)\cdot(v\rightarrow r_1))\right)\right]^i \tag{17}$$

$$= \sigma([X\cdot Y]^*) = \sigma(L) \tag{18}$$

Equations 14 and 15 are transformed as

$$\forall u \in V_1 : \mathit{Paths}(r,u) = \sigma(L)\cdot\sigma(P_1(r_1,u)) = \sigma(L\cdot P_1(r_1,u)) \tag{19}$$

$$\forall v \in V_2 : \mathit{Paths}(r,v) = \sigma(L)\cdot\sigma(X)\cdot\sigma(P_2(r_2,v)) = \sigma(R\cdot P_2(r_2,u)) \tag{20}$$

where $\mathit{Paths}(r, r_2) = \bigcup_{u\in F}\sigma(P_1(r_1, u)\cdot(u \rightarrow r_2)) = \sigma(X)$.